Journal of Parallel and Distributed Computing 157 (2021) 30–42

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

# Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems

Borja Pérez, E. Stafford, J.L. Bosque *, R. Beivide

*Department of Computer Science and Electronics, Universidad de Cantabria, Santander, Spain*

ARTICLE INFO

*Article history:*
Received 25 October 2019
Received in revised form 28 January 2021
Accepted 10 June 2021
Available online 18 June 2021

*Keywords:*
Heterogeneous systems
Load balancing
Adaptability
OpenCL
Energy efficiency

ABSTRACT

A challenge that heterogeneous system programmers face is leveraging the performance of all the devices that integrate the system. This paper presents Sigmoid, a new load balancing algorithm that efficiently co-executes a single OpenCL data-parallel kernel on all the devices of heterogeneous systems. Sigmoid splits the workload proportionally to the capabilities of the devices, drastically reducing response time and energy consumption. It is designed around several features; it is dynamic, adaptive, guided and effortless, as it does not require the user to give any parameter, adapting to the behaviour of each kernel at runtime. To evaluate Sigmoid's performance, it has been implemented in Maat, a system abstraction library. Experimental results with different kernel types show that Sigmoid exhibits excellent performance, reaching a utilization of 90%, together with energy savings up to 20%, always reducing programming effort compared to OpenCL, and facilitating the portability to other heterogeneous machines.



## 1. Introduction

In the last few years, computer architecture has been constrained by the end of Dennard's scaling. A major consequence of this is the slower growth in processor frequency. In order to find more efficient architectures, designers first resorted to increasing the number of cores per processor. Recently, the focus is on the specialization of the processing units. This tendency has fostered the advent of hardware accelerators of different kinds.

Consequently computer systems of all scales and sizes are incorporating some sort of hardware accelerators, thus becoming *heterogeneous systems*. From SoCs in mobile telephones to compute nodes on a supercomputer, they are all taking advantage of the high performance and outstanding energy efficiency of these devices. Regardless of these being GPU, TPU or FPGA accelerators, there are substantial architectural differences with the main CPU of the heterogeneous system. Interestingly, the success of these systems comes despite the fact that efficiently programming them is far from trivial.

Software development for heterogeneous systems currently relies on the *host-device* model. It dictates that the applications start running on the CPU, and purely numerical kernels are offloaded to a single accelerator in the system. Meanwhile, the CPU and other accelerators remain idle until the conclusion of the execution of the kernel. This obviously wastes the computing capability of the CPU and the other idle devices, which, to make matters worse, consume a considerable amount of energy even when idle. In OpenCL terms, a kernel is a large set of threads, or *work-items*, which are grouped in *work-groups*. These may be executed concurrently and independently in different devices. Consequently, a single kernel can be executed simultaneously on several devices, combining their computing capabilities, thus reducing energy consumption.

Several frameworks, like CUDA [29] and OpenCL [38], are available to program heterogeneous systems. They enable programmers to access the accelerators of the system, but they fail at presenting the heterogeneous system as a whole. Therefore, the programmer is left alone to face one of the most complex tasks required for an efficient use of heterogeneous systems: load balancing. Some programmers, after investing significant effort, have tackled it, like [7,19] and [32], but a generic solution to the problem is still to be found.

A further complication to the distribution of work and data, is that it must be performed in accordance to the computing capabilities of the devices, which may vary largely from system to system. A load balancing algorithm must be capable of assigning the right amount of work-groups to each device, so they all finish computing simultaneously [6,17,43]. To do this, the algorithm needs to be able to adapt both to the heterogeneity of the system and the behaviour of the applications. However, adaptiveness sometimes introduces overheads, as it increases the number of host-device in-

* Corresponding author.
*E-mail addresses:* perezpavonb@unican.es (B. Pérez), stafforde@unican.es (E. Stafford), bosquejl@unican.es (J.L. Bosque), beivider@unican.es (R. Beivide).

https://doi.org/10.1016/j.jpdc.2021.06.003

teractions. Ideally, these should be minimized while achieving an accurate load balancing, avoiding device oversubscription or underutilisation and minimizing response time and energy usage.

In order to design such a load balancing algorithm, it is also necessary to consider the behaviour of the kernels, which can be classified as *regular* or *irregular*. In the first kind, different work-groups present the same amount of computing operations. Then, for a given device, they have a constant execution time. On contrast, work-groups of irregular kernels may represent substantially different computing loads and, therefore, have an unpredictable execution time.

This article proposes *Sigmoid*, a new load balancing algorithm that achieves near-optimal performance for both regular and irregular kernels, requiring no parameters whatsoever. By utilising all the available devices in the system, it reduces response time, which can also bring a reduction in energy consumption. This is possible because Sigmoid is dynamic, enabling it to adjust to the heterogeneity of the devices. Also, since it is capable of automatically adapting to the type of kernel, it extracts maximum performance out of the system. Finally, it minimises the programming effort, as it does not require any parameter to be provided. Internally, Sigmoid starts kernel launches with some initial values and adjusts them in real time, based on measurements of the execution.

Sigmoid has been integrated in Maat [33,34], a system abstraction library that enables the transparent co-execution of a single OpenCL kernel, exploiting all the capacity of a heterogeneous system. An exhaustive experimental evaluation has been carried out in two different scenarios. First, one in which the performance of CPUs and GPUs is combined to evaluate the capability of Sigmoid to account for different kinds of devices. Second, a system with a greater number of GPUs, resembling current supercomputer nodes, to evaluate scalability. Sigmoid is compared to a heterogeneous Static [33] algorithm and two dynamics algorithms, HGuided [33] and Adaptive [6]. Experimental results show that Sigmoid can almost perfectly balance the load of all the devices in the system, regardless of the type of kernel and scenario. Performance-wise the results are close to the maximum speedups achievable for each application, and the utilisation of the system is close to 90%. Furthermore, on average, the energy consumption is reduced by up to 20%. Combining these two results, Sigmoid more than doubles the energy efficiency of the test system. Finally, these results show that Sigmoid presents good scalability in both regular and irregular kernels.

This paper contains the following contributions:

- Presents Sigmoid, a new load balancing algorithm that reduces programming effort. By combining dynamic, guided and adaptive techniques, it distributes the workload of a single kernel among all the devices of a heterogeneous system, regardless of the behaviour of the kernel.
- Explains how the Sigmoid function has been transformed to be used as part of a load balancing algorithm and how it adapts at runtime to the specific needs of the application to obtain the best results.

The remainder of the paper is structured as follows. Section 2 introduces some basic concepts that are central to the article. It is followed by a presentation of the Sigmoid algorithm in detail in Section 3. Sections 4 and 5 describe the experimental methodology and discuss the results of the experiments. Section 6 covers related literature. And finally, Section 7 gives some conclusions and future lines of work.

## 2. Background

Programming heterogeneous systems for efficiency is a complex endeavour that touches several knowledge fields. This section summarises some basic concepts that are used throughout the paper.

### 2.1. Programming heterogeneous systems

OpenCL [38] is a language and programming framework for heterogeneous systems. It favours a host-device approach to parallel programming, in which a host manages the available resources and offloads numerical kernels to different hardware accelerators.

The code that is destined for execution on the accelerators is encapsulated in usually short, data-parallel, C-like functions, which are commonly known as *kernels*. When one is offloaded to an accelerator, OpenCL launches multiple instances of the kernel code in a Single Instruction Multiple Thread (SIMT) fashion. Each instance is called a *work-item*, and the *global work size* parameter dictates how many of these items are launched.

Work-items are launched in teams so they can cooperate and synchronise with each other. Their size can be defined through the *local work size* parameter. OpenCL ensures that the work-items of each team, or *work-group*, are launched simultaneously in the same compute device. However, a device may not have enough resources to execute all the work-groups of a kernel all at once. Therefore, OpenCL states that it must be possible to execute work-groups independently. This makes them a good choice for a scheduling unit.

### 2.2. Load balancing

Maximum performance and energy efficiency can only be achieved by adequately balancing the load of a given kernel among the available computing resources. However, this is something that current frameworks do not provide. This desirable, but missing feature has been widely considered in the literature. Proposals can be classified into *task-parallel* approaches [14,16,43,40] and *data-parallel* approaches [34,17,6]. The first rely on assigning the different kernels of an application to the devices, in a way that minimises the idle time of the system. The second one, also known as *co-execution*, splits the work of a single kernel to permit several devices to cooperate in parallel. This approach is more suitable to the tasks that are usually offloaded to GPUs, as they are usually highly parallel, and therefore, it will be used in this paper.

Load balancing a single kernel on different devices rises the question of deciding the amount of work to assign to each device. Work is distributed to the devices in batches of work-groups named *packages*, the number and size of which greatly influence performance. The answer might differ depending on the type of kernel. A first approach is to divide the workload into as many adequately-sized packages as computation devices are available. This is the *Static* algorithm, which is optimal for regular kernels and if the computing speed of the devices is known.

The unpredictable nature of irregular kernels can only be correctly balanced if the work division is done during execution, using *dynamic* algorithms. These divide the workload into far more packages than devices, and have a centralised marshal that assign packages to devices upon request. This allows for a runtime adaptation to the irregularities of the kernel, thus achieving better performance. However, they suffer from a significant performance loss due to the host-device interaction required by each package.

To overcome this limitation, prior works have proposed algorithms that try to reduce the number of packages while remaining adaptive. Such is the case of the Adaptive algorithm [6], which uses small initial probe packages to attempt to obtain the computing speed of the devices and then calculate an ideal distribution of the workload. However, using small packages sometimes

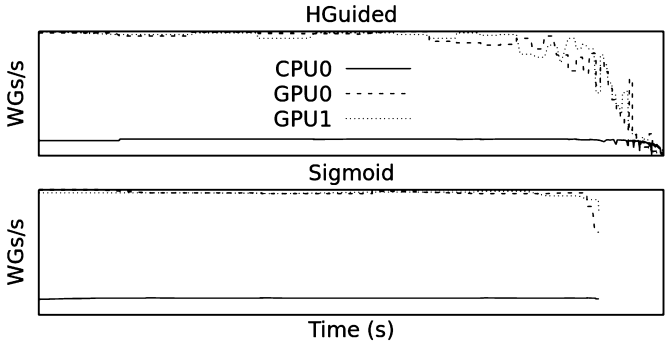


**Fig. 1.** Computing speed comparison for Binomial.

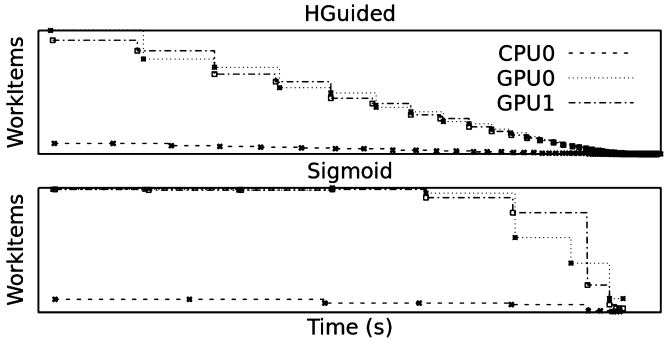


**Fig. 2.** Package size comparison for Binomial.

means wasting computing resources due to underutilization. The same problem affects other approaches like [3,28]. Moreover, experimental results presented in Section 5, show that the Adaptive algorithm produces too few packages, which ultimately compromises load balance in some cases.

The HGuided algorithm [33] takes a contrary approach. It initially launches big packages, followed by subsequently smaller ones. This reduces the amount of packages while allowing a fine grained scheduling at the end of the execution. However, it also has certain limitations.

For instance, HGuided does not fully leverage the capabilities of the devices. The top graph of Fig. 1 shows the evolution of the computing speed, expressed in work-groups per second, when a system runs the Binomial kernel using the HGuided algorithm. Notice that, for a significant portion of the execution, the devices are going slower than they could. This is because the packages it produces decrease linearly in size, as can be seen in the top graph of Fig. 2. The result is a large number of small packages near the end of the execution. This negatively impacts performance in two ways, first there is an increase of host-device interactions, and second, small packages cannot fully exploit the computational resources of the devices. This may be particularly notorious on regular kernels, which do not require adaptiveness to obtain the best performance.

The HGuided algorithm also requires certain parameters from the programmer, which strongly condition the success of the load balancing. To be accurately set, they require costly parameter sweeps for each kernel and system. Moreover, some kernels are very sensitive to these parameters, delivering highly degraded performance when using slightly off-key values. This results in dozens of tuning executions, which represent a waste of time, resources and energy. Furthermore, this is unfeasible in dynamic environments, such as a datacenter or cloud, where different applications may run on different systems, with an a priori unknown matching. This paper presents Sigmoid, a load balancing algorithm that solves the aforementioned issues.

## 3. The sigmoid load balancing algorithm

### 3.1. Overview

To solve the problems introduced in the previous section, the *Sigmoid* load balancing algorithm was conceived chasing four main objectives. First, it should successfully divide a single massively data-parallel OpenCL kernel between a set of heterogeneous devices. Second, it should evaluate the computational performance of the devices while avoiding overheads. Third, it should give good results with any type of kernel. And fourth, it should be able to be used effortlessly by the programmer. These four goals are the key to an efficient transparent use of the available resources, regardless of the underlying hardware or executed kernel.

Considering, Sigmoid is a *dynamic* and *heterogeneous* algorithm because it is able to distribute the workload among devices at run time, proportionally to their computing power. By matching the package size to the computing powers of the devices, excessively large packages are not assigned to slower devices, and the use of more powerful ones is maximised.

It is also an *adaptive* algorithm, since it is capable of modifying its operation to suit the type of kernel. It will divide regular kernels in larger packages to reduce overhead, and use smaller ones for irregular kernels, as it is impossible to predict their execution time. To do this, it continually measures the performance of the devices and tunes a number of internal parameters accordingly. Unlike other proposals, [3,6,27], this parameter tuning is performed transparently to the programmer and without any loss of performance. Thus, Sigmoid behaves equally well with both regular and irregular applications.

And finally, it is a *guided* algorithm since the package size, which is initially proportional to the computing power of each device, decreases towards the end of the kernel execution. This satisfies several goals. Using large packages at the beginning of the execution reduces the overhead. And decreasing the size of the last packages improves the accuracy of the load balancing. In addition, this keeps the utilisation of the devices nearly constant throughout the execution, which has a strong impact on performance.

### 3.2. Algorithm description

As Sigmoid takes a dynamic approach to load balancing, it first launches an initial package to each of the available devices and then waits until any of them completes their execution. The packages are sized in accordance to an initial computing speed, based on the GFLOPs reported by the specs of the devices. When one finishes the execution, if there is pending work, a new package is generated and issued to the idle device. To improve the load balance, the size and response time of completed packages are analyzed to tune the internal parameters of the algorithm throughout the execution of the kernel.

How the package size evolves throughout the execution of a kernel is key to an efficient load balancing. This is because package size poses a dilemma: smaller packages garner greater adaptiveness, but also greater overhead. Moreover, computing speed is sometimes correlated with the work quantity offloaded to a device, so small packages often lead to suboptimal performance, that is not representative of the actual capabilities of the hardware. The importance of this phenomenon has been already addressed in [3]. Thus, a good load balancing algorithm will attempt to keep computing speeds and, consequently, package sizes as high as possible, while not compromising adaptiveness.

To calculate successive package sizes, Sigmoid relies on a function that issues big packages for most of the execution and, gradually, smaller ones at the end, which reduces overheads while maintaining adaptiveness and keeping device utilisation high. This

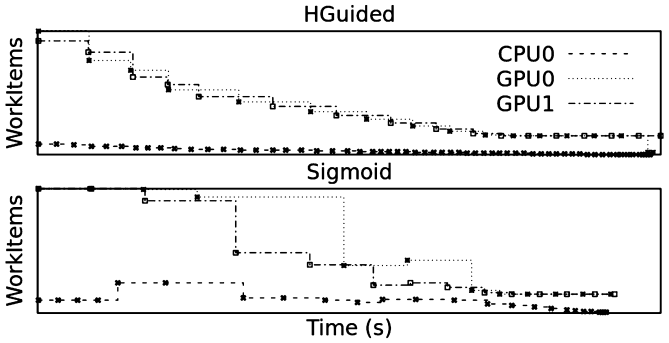


**Fig. 3.** Package size comparison for BM3D.

is depicted in Figs. 1 and 2, which compare the computing speed and package size evolution when executing the Binomial benchmark with the HGuided and Sigmoid algorithms. Note how Sigmoid generates fewer and bigger packages, maintaining computing speed high for longer. The decrease rate of the package size is adjustable through an internal parameter, which varies depending on the behaviour of the kernel.

Sigmoid uses the size and response time of executed packages to detect if the kernel is irregular, and adjust the decrease rate to generate smaller packages, if more adaptiveness is required. The algorithm also automatically identifies an adequate minimum package size that strikes a balance between adaptiveness and performance, and calculates the computing speed of the devices to avoid imbalances. The result is an algorithm that adapts to the behaviour of kernels. This is shown in Fig. 3, which compares the package sizes generated by HGuided and Sigmoid for an irregular kernel. Again, HGuided generates linearly decreasing package sizes, although a certain distortion can be appreciated due to irregularity. Sigmoid, in turn, uses variable package sizes to adapt to the kernel. This can be seen in the humps near the end of the execution of GPU0 and GPU1, which account for computing speed variations associated to package workload differences. An exhaustive package size evolution analysis has been carried out for every evaluated application, however it has been left out due to space limitations. Figs. 2 and 3 have been found representative of the behaviour of regular and irregular kernels respectively. A high level description of the algorithm can be seen in Algorithm 1. The following sections explain the different internal parameters and functions of the algorithm and will refer to Algorithm 1.

**Algorithm 1:** Sigmoid algorithm.

**Input:** The number of work-groups $WG$, a set of $N$ devices with $S_j$ default computing speeds

$x \leftarrow G$ *(Number of remaining work-groups)*
$k \leftarrow k_r$ *(Slope internal parameter)*
**for** $j \leftarrow 1$ ***to*** $N$ **do**
  $oc_j \leftarrow$ Occupancy lower bound for device $j$
  $c \leftarrow package\_size(j, x)$
  Schedule $c$ work-groups to device $j$
  $x \leftarrow x - c$
**end**
**while** $x > 0$ **do**
  $(j, c, t) \leftarrow$ Wait for any device
  $S_j \leftarrow$ Average of the last 3 computing speeds $(\frac{c}{t})$
  $\sigma_{S_j} \leftarrow$ Standard deviation of last 3 computing speeds
  **if** $\frac{\sigma_{S_j}}{S_j} > 0.25$ **then**
    $k \leftarrow k_i$
  **end**
  $c \leftarrow max(package\_size(j, x), ptS_j, oc_j)$
  Schedule $c$ work-groups to device $j$
  $x \leftarrow x - c$
**end**

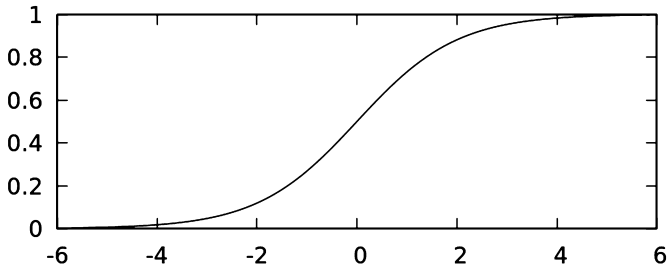


**Fig. 4.** Representation of the logistic function for $L = 1$, $k = 1$ and $x_0 = 0$.

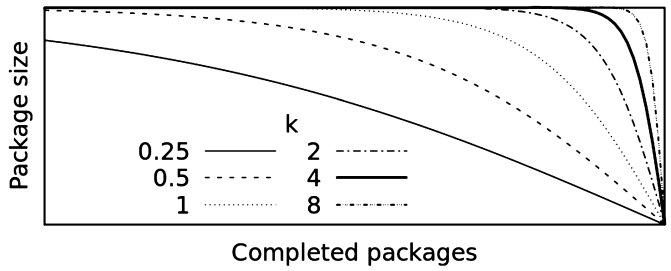


**Fig. 5.** Evolution of the package size for different $k$ values.

### 3.3. The logistic function for load balancing

The logistic function is used to model processes that appear in many fields, ranging from biology to medicine, and commonly used as machine learning activation function [13,5]. This function, conveniently transformed, is the foundation of the Sigmoid load balancing algorithm. It is defined by the following equation and a graphical representation is shown in Fig. 4.

$$logistic\_function(x) = \frac{L}{1 + e^{-k(x - x_0)}} \tag{1}$$

To apply the function to the load balancing problem, variable $x$ will represent the amount of remaining work-groups. Consequently, it will be monotonically decreasing and take values between $G$, the total number of work-groups that have to be processed, and 0. As $x$ will always be positive, parameter $x_0$ is eliminated. The maximum value of the function, $L$, will represent the size of the largest package. It can be seen in Fig. 4 that for $x = 6$ the function yields a value close to the asymptotic maximum. To obtain a Sigmoid curve that captures the desired behaviour for the package size decrease rate, which is shown in Fig. 5, Sigmoid will only use the function in the $[0, 6]$ interval. This produces and even decrease in the package size, allowing for adaptiveness and preventing steep size changes. Variable $k$ is the slope of the curve that is internally calculated by Sigmoid to modify the rate at which package size decreases. The following lines show in detail how the internal Sigmoid function, represented in Fig. 5, is derived from the logistic function.

Let $G_r$ be the number of remaining work-groups. As the aim is to obtain a function $f$ that produces decreasing package sizes as the execution of the kernel progresses, $G_r$ will be used as the $x$ in the logistic function. However it will be normalised to the total number of work-groups $G$ and scaled to 6 to map it to the $[0, 6]$ interval.

$$f(G_r) = \frac{L}{1 + e^{-k\frac{6(G_r)}{G}}} \tag{2}$$

So far, since $0 \leq G_r \leq G$, then $f(G_r)$ returns values between $\frac{L}{2}$ and $L$. It is necessary to transform this to appropriate package size values. First, the range of the function is mapped to the $[0, L]$ interval by multiplying by 2 and subtracting $L$.

$$f(G_r) = \frac{2L}{1 + e^{-k\frac{6(G_r)}{G}}} - L \tag{3}$$

And second, it is necessary to find $L$, which is the maximum value the function will take, and will be used as the size of the first package scheduled by the algorithm. The chosen value is $\frac{G}{2N}$, where $N$ represents the number of available devices. This is equivalent to the base chunk size commonly used by the OpenMP Guided algorithm [9].

$$f(G_r) = \frac{2\frac{G}{2N}}{1 + e^{-k\frac{6(G_r)}{G}}} - \frac{G}{2N} \tag{4}$$

To account for the heterogeneity of the system, a correction based on the computing speed of each device is added. The speed $S_i$ is defined as the number of work-groups that device $i$ can compute per second. Similarly, the aggregated computing speed of the system is represented by $S_T$. And given the number of scheduled work-groups $G_r$, the size of the next package for device $i$ is as follows.

$$\begin{aligned} Pack\_size(i, G_r) &= f(G_r)\frac{S_i}{S_T} \\ &= \left(\frac{2\frac{G}{2N}}{1 + e^{-k(6\frac{G_r}{G})}} - \frac{G}{2N}\right)\frac{S_i}{S_T} \\ &= \frac{1 - e^{-k(6\frac{G_r}{G})}}{1 + e^{-k(6\frac{G_r}{G})}}\frac{G}{2N}\frac{S_i}{S_T} \end{aligned} \tag{5}$$

This function is used to obtain the size of the packages, but to avoid excessive overheads the size is not allowed to drop below two lower bounds. How Sigmoid automatically obtains these two values and the slope of the function is explained next.

### 3.4. Automatic parameter tuning

The above expression requires a series of parameters. Some of them are known beforehand, like the number of devices $N$, the total number of work-groups $G$ or the number of remaining work-groups $G_r$. Others must be computed and updated as the kernel execution progresses. Such is the case of the computing speed of each device $S_i$ or the slope of the Sigmoid curve. Moreover, a minimum package size has to be selected in order to avoid a large number of small packages at the end of the execution, as they would increase the host-device interaction overhead and reduce the computing speed of the devices due to their small size. The automatic update of these parameters is what allows Sigmoid to autonomously adapt itself to different kernel behaviours. In this section we will explain how these parameters are obtained.

The computing speed of the devices is used to tailor the amount of work to be distributed according to the capabilities of the receiving device. These values can be easily computed at runtime by monitoring the kernel execution. However, computing speeds are kernel dependent. Consequently, for the first packages of a kernel, speed information will not be available, so an approximation is necessary. As an estimation, the nominal GFLOP values reported by the hardware vendors, are initially used to calculate the relative speed of the devices. These values may not accurately represent the capabilities of the devices for the current kernel, but an approximate speed estimation at the beginning of the kernel execution does not have a large impact on performance. It is at the end of the execution when accurate speeds are required, and by then the algorithm will have refined these throughout the duration of the whole kernel. This is done by measuring the time that each package takes to execute and calculating its speed in work-groups per second. To reduce the influence of work bursts, that may not be representative of the behaviour of the whole workload, the average speed shown by the last packages launched to each device is used to update the values used by Sigmoid. This is shown in line 11 of Algorithm 1. However, keeping track of a very long package history might have a negative effect on adaptiveness. This is because, the longer the history, the longer it will take Sigmoid to converge to the new speed after a change in the behaviour of the workload. It was experimentally found that three packages strike a balance between adaptiveness and over-sensitivity. Consequently, Sigmoid will converge to the speed shown by a device once three packages have been executed. In the case of irregular applications, the algorithm will adapt accordingly to speed changes.

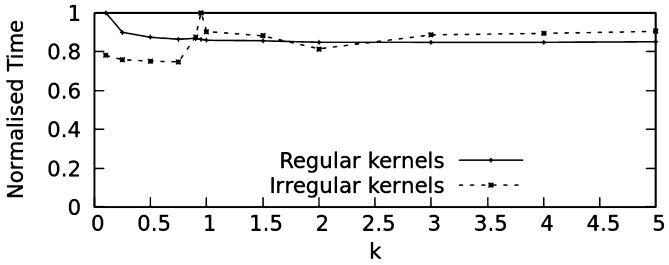


**Fig. 6.** Influence of $k$ on regular and irregular kernels (lower is better).

The slope of the Sigmoid curve, represented by $k$, controls the rate at which the package size decreases and, ultimately, the degree of adaptiveness of the algorithm. As shown in Fig. 5 a greater $k$ gives a steeper curve, producing fewer and bigger packages that limit adaptiveness at the end of the execution. This situation suits regular kernels, which do not require adaptiveness, and benefit from the reduced overhead. Irregular kernels, on the other hand, will require a smaller $k$ value that will increase adaptiveness, at the cost of a higher overhead. To choose adequate values of $k$ a set of executions of all the kernels used in the evaluation (Section 5) was done using different values. The results of these experiments showed that it is sufficient to use two different $k$ values to achieve good results in both regular and irregular kernels. The values thus selected have been, $k_i = 0.5$ for irregular kernels and $k_r = 2$ for regular ones. The reason for this choice is that these values deliver good overall performance and belong to stable intervals, in which small $k$ differences do not represent great performance variability. Fig. 6 shows an example of this behaviour for two representative kernels: Mandelbrot as regular, and Ray as irregular. The chosen values for $k_i$ and $k_r$ are expected to provide good performance for other kernels. Nevertheless, for strictly optimal performance, a slight adjustment of these parameters might be necessary when executing other kernels.

In order to apply the correct $k$ value it is necessary to determine which type of kernel is being executed. When a kernel is launched, it is regarded as regular until proven otherwise. This avoids penalizing regular kernels and should not affect irregular ones, as adaptiveness is most necessary near the end of the execution. Consequently, packages are initially distributed using $k_r$. Irregularity is defined by a variability in the time taken to execute two equally-sized chunks of work. Therefore, to switch between $k_r$ and $k_i$ the variability of the computing speed for each device is analysed. To do so, Sigmoid considers the standard deviation of the speed of the last three packages ($\sigma_{S_i}$) on the current device $i$. If the ratio between this value and the average speed $S_i$ rises above a given threshold $d$, the kernel is deemed irregular and $k_i$ is used. Note that once a kernel is considered irregular, $k_i$ will be used for the remainder of its execution. This is shown in lines 12-15 of Algorithm 1. This is due to the fact that some irregular kernels may have regions of regular behaviour, in which the standard deviation ratio might drop below the threshold. However, there is no guarantee that irregularity will not be present again near the end of

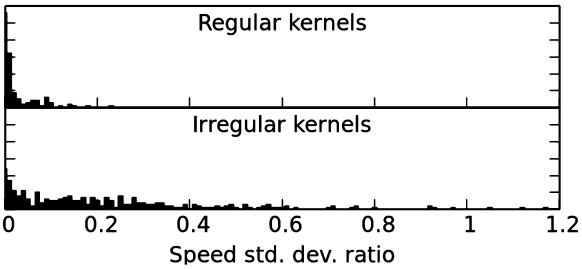


**Fig. 7.** Histogram of the number of packages for regular and irregular kernels with respect to the speed variability percentage.

the execution, with few opportunities to react. Therefore using $k_r$ again in an irregular kernel would greatly harm performance. Nevertheless, to verify this hypothesis, tests were carried out using a version of Sigmoid that falls back to $k_r$ if regularity is detected. This version caused an average performance loss of close to 10%.

To adequately set the value for the threshold $d$, an analysis of performance variability in regular and irregular kernels was necessary, as even regular kernels present certain performance differences due to several factors, such as cache effects or contention. To decide the threshold $d$, the value of $\sigma_{S_i}$ for each package executed on all the evaluated kernels has been studied. Fig. 7 depicts histograms of the standard deviation ratio $\frac{\sigma_{S_i}}{S_i}$ for regular and irregular kernels. As can be seen, packages obtained in regular kernels present a maximum performance variability of around 20%, while differences are much greater for irregular ones. Considering this, $d$ has been set to 0.25 to avoid misidentifying regular kernels.

The purpose of applying lower limits to the size of the packages generated by Sigmoid is twofold. First, it strives to contain the excessive overhead that is inherent in small packages. Second, it guarantees that package sizes do not decrease to a point in which the resources of the devices are not fully used. However, these factors should not be at odds with adaptiveness or induce imbalance in order to keep utilisation high.

Targeting the first mentioned purpose implies a risk, because avoiding overheads by increasing package size might generate imbalances, arising from the time difference among the terminations of the last package scheduled to each device. In a worst case scenario, this imbalance might represent the whole execution time associated to the last package. Accounting for this, a maximum imbalance coefficient $p$ is defined. This represents the maximum imbalance that will be generated by Sigmoid in the aforementioned worst case scenario. Then, $p$ is used in the following equation, together with the current execution time $t$ and the average device speed $S_i$, to obtain a minimum package size that limits overheads but does not generate significant imbalance.

$$Minimum\_package\_size = pt\,S_i$$

Guided by the benchmarks used in this paper, $p = 0.05$ has been chosen, which does not cause excessive overheads and avoids imbalance. Conceptually, this means that, at the current speed, the execution of a package launched to device $i$, with a size calculated using the equation, will represent 5% of the current total execution time. Equivalently, in a worst case scenario, at most 5% of the current runtime will be spent in an imbalanced execution, with only one device computing and the rest idling.

The second lower bound for the package size guarantees that the devices are fully used. For GPUs, the algorithm implements the equations of the CUDA Occupancy Calculator, which is part of the CUDA Toolkit since version 4.1. These, take the number of registers and the amount of shared memory required by a kernel, which are values that can be obtained from the OpenCL compiler, and calculate its maximum occupancy and the number of work-groups per multiprocessor required to reach it. The latter value multiplied by the number of stream multiprocessors in the GPU, which can also be queried from OpenCL, is the minimum number of work-groups to achieve maximum occupancy. CPUs usually show a much more regular performance on the number of work-groups than GPUs, so on CPUs this lower bound is set to one work-group per CPU core.

As a result, the package size that will be selected will be the maximum of these two lower bounds and the value obtained using the Sigmoid function. This is shown in line 16 of Algorithm 1.

## 4. Experimental methodology

This section describes some details of the experimental setup used to evaluate Sigmoid. This includes a description of the load balancing algorithms used for comparison, test hardware, the choice of benchmark kernels and measurement tools.

### 4.1. Load balancing algorithms

In order to evaluate the improvements of Sigmoid, the following well-known load balancing algorithms were considered in the experiments.

Static algorithm [33]. This classic algorithm divides the kernel in as many packages as devices are available in the system. The size of each package is proportional to the relative computing speed of the device that will execute it. This algorithm minimizes the overhead, since only one package is sent to each device. Thus, for regular kernels this is a priori the best choice. However, the algorithm requires the computing speed of the devices as input parameters and it performs badly with irregular kernels.

HGuided algorithm [33]. This algorithm aims to reduce the overheads associated with host-device interactions while retaining the best adaptiveness. Like in the *guided* method from OpenMP [9] the size of the packages starts being large and diminishes as the execution progresses. As parameters, it requires the different computing speeds of the devices and a minimum package size. Two different versions have been considered. The first one, labelled *BestHG*, uses optimal parameter values for each kernel and system, obtained after an exhaustive parameter sweep. The second, *DefHG* represents an effortless usage of HGuided. To be fair, the same parameters initially used by Sigmoid have been selected: the nominal values for the computing speeds, and the values reported by the CUDA occupancy calculator for the minimum package size.

Adaptive algorithm [6]. This algorithm was proposed for a two device scenario as a dynamic technique that requires no training and responds automatically to performance variability. But, the implementation used in this paper is an extension of the original algorithm for an arbitrary number of devices introduced in [33]. It proceeds by first launching small probe packages to the devices and then using their execution times to predict an ideal static work partitioning for the remaining work. The amount of probe packages per device, their size and growth rate are programmer defined parameters. The authors suggest a set of parameters that deliver good performance, which are used in this work. However, it was experimentally found that the suggested size of the first probe package was too big and the slowest device did not finish enough packages for the adaptive distribution to begin before the other devices completed the rest of the work. Consequently, a smaller first package has been used. Similar algorithms in the bibliography are [27,28,3].

From the aforementioned algorithms, Static and BestHG take parameters, which have a strong impact on performance. Often, these have different optimal values for each kernel and system

**Table 1**
Parameters for each Benchmark.

| Benchmark | Type | Problem size | Local work size | GPU computing speed | Minimum size |
|---|---|---|---|---|---|
| Binomial | Regular | 2048000 | 256 | 7.28 | 380 |
| Gaussian | Regular | 8000×8000 81×81 | 128 | 13.77 | 1000 |
| Mandelbrot | Regular | 20480×20480 | 256 | 5.88 | 400 |
| NBody | Regular | 51200 | 128 | 7.33 | 400 |
| Taylor | Regular | 800×800 | 128 | 2.06 | 280 |
| Aho | Irregular | 1536000 | 64 | 8.20 | 200 |
| BM3D | Irregular | 800×800 | 64 | 2.28 | 150 |
| Rap | Irregular | 1024×1024 | 64 | 4.26 | 400 |
| Ray | Irregular | 12000×12000 | 64 | 7.70 | 380 |

configuration. As a consequence, it was necessary to individually tune them to each benchmark, in order to obtain their best possible results. This process meant performing thousands of executions of the benchmarks in time-consuming parameter sweeps, which would be required for any new benchmark or hardware configuration.

### 4.2. Test platform and benchmarks

Experimentation has been carried out using two different machines. The first one, labelled *Batel*, has two CPUs, two GPUs and 16 GBs of DDR3 memory. Both the CPUs and GPUs take part in co-execution. The CPUs are Intel Xeon E5-2620, with six cores that can run two threads each at 2.0 GHz. The CPUs are connected via QPI, which allows OpenCL to detect them as a single device. Therefore, throughout the remainder of this document, any reference to the CPU includes both Xeon E5-2620 processors and all their cores. A load balancing scheme that consider each individual core separately, using *device fission*, was evaluated. However, it was found to deliver worse performance than the work distribution obtained when the OpenCL driver is in charge of distributing work-groups to each of the CPU cores. This could be attributed to overheads or cache affinity issues, as device fission provides no means to identify the placement of each of the returned cores in the memory hierarchy.

The GPUs are NVIDIA Kepler K20m with 13 SIMD lanes (or SMs in NVIDIA terminology) and 5 GBytes of VRAM each [30]. These are connected to the system using independent PCI 2.0 slots. For performance and energy experiments, the baseline system uses a single GPU, but the static energy of the unused devices, which are idle but still consuming, is considered. This accounts for the fact that current HPC systems often incorporate several accelerators, which, if unused, would represent a considerable energy waste.

The second machine, labelled *Hydra*, has been used to evaluate the scalability of Sigmoid as compared to that of the other algorithms. It has four NVIDIA GeForce GTX TITAN Black GPUs, each one having 15 SIMDS lanes and 6 GB of VRAM. Note that the CPU does not take part in co-execution in the tests that use this system. This is to better analyze the scalability itself, by evaluating the behaviour of the algorithms when the number of identical devices increases.

Nine kernels have been chosen for the experiments. Five of which exhibit regular behaviour. Binomial (Bin) generates binomial lattices, useful for option pricing in financial software. Mandelbrot (Man) implements a blocked algorithm to compute a Mandelbrot set. NBody (Nbo) simulates a dynamic system of particles, used in many physics applications. Gaussian (Gau) calculates the Gaussian blur of an image, commonly found in image and video processing software. The last regular kernel is Taylor (Tay), which performs a bi-dimensional Taylor approximation for a set of points. The other four kernels are irregular. Aho is an implementation of the Parallel Failureless Aho-Corasick (PFAC) string matching algorithm, commonly used for protein sequencing [24]. BM3D (BM3) implements one of the filters of the BM3D image denoising algorithm [10]. Rap is an implementation of the Resource Allocation Problem [1]. There is a certain pattern in the irregularity of RAP, because each successive package represents a bigger amount of work than the previous. Finally, Ray Tracing which renders realistic images by calculating the light that reaches each pixel by modelling light rays. Two different scenes of similar complexity but with different object distribution, (Ray1 and Ray2), have been defined. It will be shown later (Section 5) that changing the input data, the behaviour of the application varies wildly.

Table 1 shows some parameters for each kernel and load balancing algorithm. The “GPU computing speed” and “Minimum size” columns indicate the values for the parameters required by the Static and HGuided load balancing algorithms. The former represents the computing speed of the GPU relative to that of the CPU. The latter is the minimum package size generated by the HGuided algorithm, expressed in work-groups. The local work size has been set to maximise the performance of the fastest device, namely the GPU. The reason for this is that almost no performance difference was detected when varying the local work size for the CPU.

### 4.3. Energy measurements

To measure the energy consumption of the system it is necessary to take into account the power drawn by each device. Modern computing devices allow applications to monitor their functionality and performance. However, the power measured is associated to the device and not the kernel or process in execution. Together with the fact that it is impractical to add measurement code to all the test applications, this led to the development of a power monitoring tool named *Sauna*. It takes a program as its parameter, and is able to periodically query all the devices for power measurements throughout the execution of the program.

A significant amount of thought went into the conception of Sauna; the fact that it had to monitor several devices meant that it had to adapt to the particularities of each one while giving consistent and homogeneous output data. This started with the different APIs provided to perform these measurements. For the Intel CPUs, recent versions of the Linux kernel provide access to the *Running Average Power Limit (RAPL)* registers [35], which provide accumulative energy readings. On contrast, NVIDIA provides the *NVIDIA Management Library (NVML)* [31] that gives instant power measurements. Naturally, Sauna had to be able to convert between the two magnitudes. A particularly interesting aspect of the development process of Sauna was studying the impact of the sampling frequency. In order to keep the program simple, it was necessary to use a single sampling period for all devices. Given that the power variations would be similar across devices, the idea seemed feasible.

To find the best frequency, a series of experiments were made for each device in *Batel* (Section 4.2). It was observed that each device reacted differently to the sampling frequency. The RAPL measurements grew with large frequencies. And more surprisingly, the NVIDIA devices slowed down noticeably when the sampling frequency was above a given threshold. This actually meant that

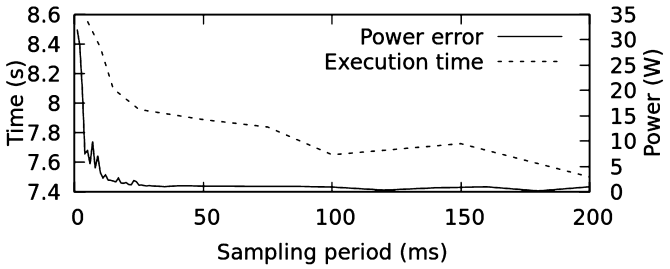


**Fig. 8.** Impact of sampling period on power measurement and kernel execution time.

the kernel running in the device took longer to complete. Fig. 8 shows these effects as the magnitude of the variation of the measured CPU power depending on the sampling rate, together with the execution time of a Binomial kernel on a NVIDIA GPU. These graphs suggest adopting a low frequency, however, if the sampling period is too long, fast power spikes that may appear under irregular loads could be missed, leading to inexact results. It was decided to use a sampling period of 45 ms since the increase in the execution time and the power error is restricted to 6%.

## 5. Experimental evaluation

This section presents the experimental results obtained on the test systems when running the different benchmarks, as described in Section 4. These experiments aim to answer the following questions:

- How well does Sigmoid balance the workload across different heterogeneous devices?
- What is the performance of Sigmoid for both regular and irregular kernels?
- Is well-balanced co-execution capable of improving the energy consumption of a heterogeneous system?
- How does Sigmoid scale when the number of devices increases?

Each of the following sections answer one of the aforementioned questions, comparing the results achieved for Sigmoid with other load balancing algorithms.

### 5.1. Load balance

The first metric considered in this analysis is the *Load Balance* which is shown in Fig. 9, for the *Batel* system. For a given execution, it is defined as the ratio of the response time of the first device to conclude its work and that of the last. The ideal value for this metric is one, meaning that all devices finished simultaneously and the maximum utilisation of the machine was reached.

Sigmoid reaches perfect load balance in six out of the ten benchmarks. Compared to the other algorithms, it obtains the best load balance, except in Rap and NBody where it is slightly worse than BestHG and DefHG. Looking at the geometric mean, Sigmoid boasts almost perfect load balance (0.97) closely followed by BestHG (0.94). Recall that the parameters of BestHG are optimal, obtained from a time-consuming sweep. Regarding the other algorithms, Static performs well in regular benchmarks but, as is expected, performs poorly in irregular ones. This is a consequence of the nature of these kernels, that make it very difficult to devise a fair load distribution before the actual execution. In the same way, Adaptive shows good results for regular kernels (except NBody) but no so satisfactory for irregular ones.

### 5.2. Performance

To give an idea of performance, Fig. 10 shows the speedups reached by the different benchmarks in the *Batel* system, compared to the baseline scenario that only uses one GPU. The test system is composed of $N = 3$ devices, but since they do not have the same computing power, the speedup is never going to reach 3. Table 2 summarises the maximum speedup $S_{max}$ each benchmark can reach. These values were derived from the response time $T_i$ of each device, in relation to the time for the fastest device in the system, as shown in Equation (6). This is, the maximum speedup is the addition of the relative performance of each device with respect to the fastest of them. The obtained values are also represented in Fig. 10 as a horizontal line above the bars of each benchmark.

$$S_{max} = \sum_{i=1}^{N} \frac{T_{min}}{T_i} \tag{6}$$

Looking at the geometric mean of the speedups shown in Fig. 10, it can be seen that Sigmoid gives the best performance. It is 22% better than Static and 3% better than BestHG. Regarding the mean for regular and irregular kernels separately (not depicted), Sigmoid obtains the same performance (99%) as the Static for regular benchmarks and is even slightly better than BestHG for irregular. In short Sigmoid delivers the best overall performance and also equals the performance of the best alternative for both regular and irregular workloads. When compared to the speedup of the other effortless algorithms, Sigmoid also excels. It is 20% better than Adaptive and 7% better than DefHG.

Regarding each benchmark individually, Sigmoid gives the best performance in all except NBody, Taylor and Rap. Despite that Sigmoid attains the best load balance results in NBody and Taylor, using the optimal parameters with Static obtains a better speedup. Since these benchmarks have a very low computation-communication ratio, the overhead increases when the workload is subdivided in more packages than devices. With Rap, Sigmoid delivers the second best performance. This is because the minimum package size that guarantees efficient device use, generates a slight imbalance at the end of the execution. Regarding the effortless algorithms, Sigmoid delivers the best performance in all the applications but Taylor, in which it is only marginally surpassed by Adaptive.

The gap between the measured and the theoretical maximum values is a consequence of the extra communication overhead that comes from having more than one device. This is more notorious in applications in which the data can not be divided and must be replicated (NBody) or when the ratio between the computation and communication times is small (Mandelbrot, RAP).

As discussed above, one of the advantages of Sigmoid is that it tries to reduce the number of packages, as each implies interaction between the host and a device, while maintaining adaptability. This can be seen in Table 3, which depicts the number of packages generated by each algorithm excluding Static, which would always generate as many packages as devices. Adaptive produces almost the same amount of packages for all benchmarks. This translates into good results in very regular benchmarks, as overheads are reduced. However, it fails in irregular ones, to which it cannot adapt. As for the HGuided algorithms, both versions generate huge amounts of packages, many more than the rest, although slightly less in BestHG thanks to the tuning of the parameters. This occurs even in regular benchmarks, like Binomial, which do not take advantage of adaptability. This causes two damaging effects. On the one hand, it notably increases overheads. On the other hand, a large number of packages are excessively small and do not fully

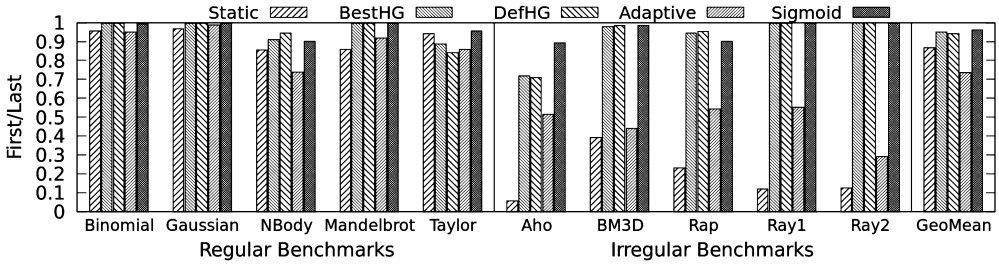


**Fig. 9.** Load balance of each device for all algorithms and benchmarks in the heterogeneous system.

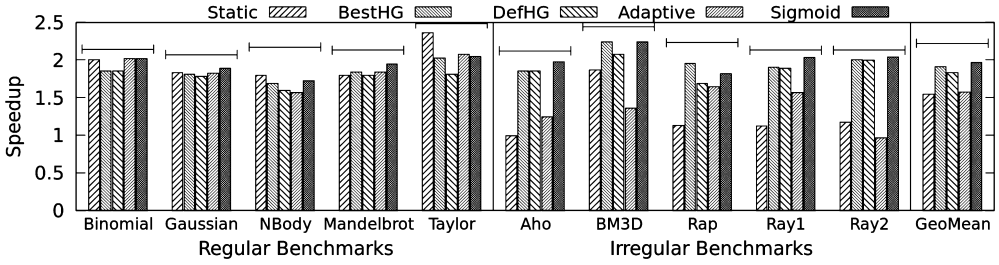


**Fig. 10.** Speedups of the benchmarks with the different algorithms in the heterogeneous system.

**Table 2**
Maximum speedup for the different benchmarks.

| Benchmark | Binomial | Gaussian | Mandelbrot | NBody | Taylor | Aho | BM3 | RAP | RAY |
|---|---|---|---|---|---|---|---|---|---|
| Max. Speedup | 2.14 | 2.07 | 2.17 | 2.13 | 2.48 | 2.12 | 2.44 | 2.23 | 2.13 |

**Table 3**
Number of Packages generated by each load balancing algorithm and benchmark.

| Benchmark | Binomial | Gaussian | Mandelbrot | NBody | Taylor | Aho | BM3 | RAP | Ray1 | Ray2 | Average |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Adaptive | 15 | 19 | 13 | 15 | 15 | 11 | 13 | 15 | 14 | 13 | **14.30** |
| DefHG | 445 | 105 | 25 | 175 | 51 | 828 | 92 | 79 | 389 | 291 | **248.00** |
| BestHG | 307 | 84 | 14 | 119 | 31 | 707 | 47 | 46 | 288 | 185 | **182.80** |
| Sigmoid | 28 | 31 | 17 | 27 | 20 | 13 | 28 | 20 | 40 | 32 | **25.60** |

take advantage of the capacity of GPUs. Finally, it should be noted that Sigmoid generates a much smaller number of packages, thus reducing the overhead with respect to HGuided. At the same time, it maintains adaptability according to the needs of each benchmark, surpassing Adaptive in this regard.

### 5.3. Energy consumption

Nowadays, performance is not the only figure of merit used to evaluate computing systems. Their energy consumption and efficiency are also very important. Fig. 11 gives an idea of the energy saving obtained by taking full advantage of all the compute devices in the *Batel* heterogeneous system. Contrasting with the baseline system that only uses one GPU, while the other devices are idle but still consuming. Therefore, the figure shows, for each benchmark, the energy consumption of each algorithm normalised to the baseline consumption. In this graph, less is better, and bars over one indicate that the whole heterogeneous system consumes more energy than the baseline.

The energy measurements are strongly correlated to the performance of the algorithms. Observing the geometric mean, it can be seen that Sigmoid gives the best results, followed by BestHG, presenting energy savings of 9% and 7% respectively. Looking closely at some benchmarks, the other algorithms can consume significantly more energy than the baseline (Static and Adaptive in irregular benchmarks). Even DefHG and BestHG do not reach any improvement in Binomial and Gaussian. Interestingly, the only algorithm that always improves the baseline consumption is Sigmoid. The use of more devices logically increases the instantaneous power at any time. But, since the total execution time is reduced, the total energy consumption is also less. This saving is further improved by the fact that idle devices still consume energy, so making all the devices contribute work is beneficial. Notice that, of the effortless algorithms, Sigmoid attains the lowest energy consumption while Adaptive presents and overall energy consumption greater than the baseline.

Another interesting metric is the energy efficiency, which combines performance with energy consumption. Fig. 12 shows the Energy Delay Product (EDP) [8], of the algorithms normalised to that of the baseline. Since this is a combination of the two above metrics, the relative advantage of the different algorithms is maintained. The geometric mean shows that with this metric all algorithms are advantageous, Sigmoid giving the best results with a 54% improvement. BestHG also gives good results (52%) since its parameters have been optimised. These two algorithms give good results in all the benchmarks, while the remaining algorithms exhibit a strong variability, in some cases even with normalised EDP values over one.

In summary, these results prove that co-execution improves the energy consumption of heterogeneous systems, in addition to their performance, as shown in the previous section.

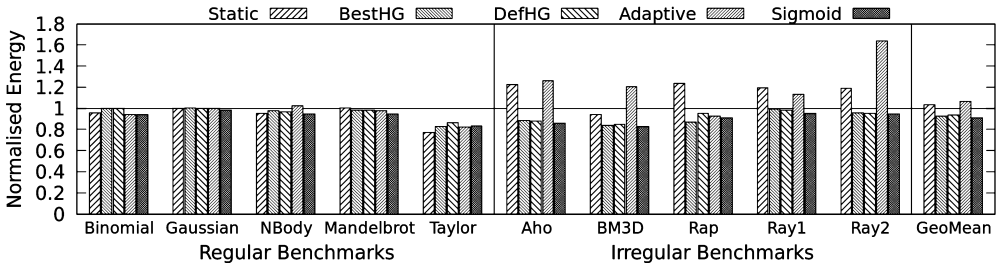


**Fig. 11.** Energy consumption of the benchmarks with the different algorithms normalised to the baseline in the heterogeneous system.

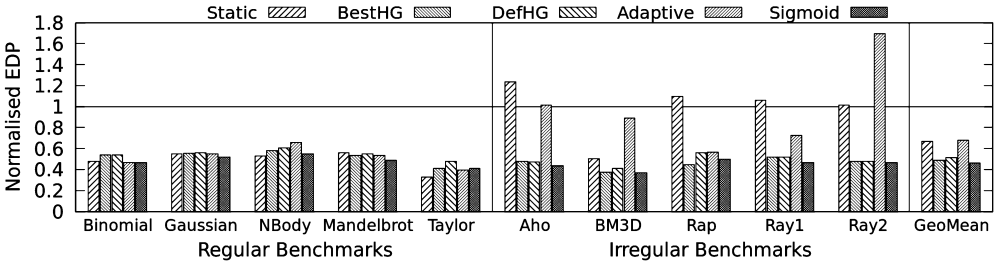


**Fig. 12.** EDP of the benchmarks with the different algorithms normalised to the baseline in the heterogeneous system.

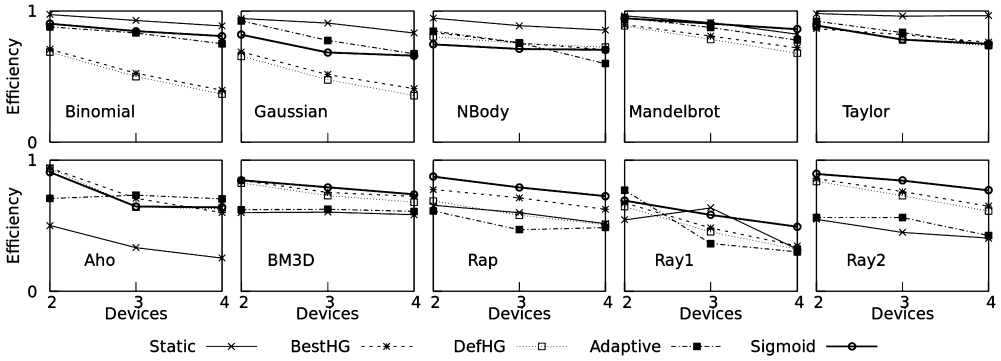


**Fig. 13.** Efficiency of the different algorithms executing the benchmarks on a homogeneous system.

### 5.4. Scalability

The last set of experiments was developed in *Hydra*, a system with four identical GPUs. It evaluates the *strong scalability* of the load balancing algorithms. Therefore, the same problem size has been used for all the experiments, while the number of devices increases from 2 to 4. For a better comparison, the metric used to evaluate scalability is the efficiency, defined as the ratio of the achieved speedup to the number of devices that have been used. Consequently, perfect scalability implies constant efficiency, meaning that the same performance per device is obtained regardless of their number. This is usually not the case and, as shown in Fig. 13, efficiency drops in all cases. The smaller the drop, the better the scalability of the algorithm.

These results show that Sigmoid is the only algorithm that scales well for both regular and irregular benchmarks. Regarding the rest of the algorithms, Static scales very well in regular algorithms, but it has serious problems in irregular ones, such as Aho and Ray1, where it scales well between 2 and 3 devices, but drops strongly with 4 devices. Adaptive also scales well on regular benchmarks, excluding NBody with 4 devices, but behaves very poorly on irregular ones. Finally, both BestHG and DefHG, have an erratic behaviour in regular benchmarks, with good scalability for NBody, Mandelbrot and Taylor, but scaling badly for Binomial and Gaussian.

Both versions of the HGuided also show the same behaviour for irregular benchmarks: good scalability for Aho, BM3D and RAP, but very poor for Ray1 and Ray2. In sum, out of the evaluated algorithms, only Sigmoid delivers uniform scalability results, regardless of the behaviour of the workload: regular or irregular.

Currently, some servers for supercomputers have even more than four GPUs, a tendency that is expected to grow in the near future. For this reason, and based on the data obtained in these experiments, an estimation of the *weak scalability* of Sigmoid has been made with up to 16 devices using the Law of Gustafson [15]. Evaluating strong scaling for such a high number of devices would require problem sizes that cannot be executed on just four devices due to memory constraints. Fig. 14 shows how efficiency evolves by increasing both the number of devices and the size of the problem, so that the workload per device is always constant. As depicted, weak scaling for Sigmoid is almost perfect, according to Gustafson's Law estimates.

In conclusion, Sigmoid achieves almost perfect load balancing, delivering excellent energy and performance results of all the considered effortless load balancing algorithms. Moreover, it delivers better overall performance than the load balancing algorithms that require parameters, it also equals the results of the best algorithms for regular and irregular kernels individually. Finally, it is the only one with good scalability in both regular and irregular applications.

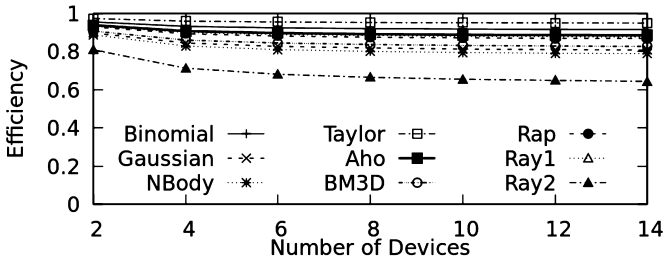


**Fig. 14.** Estimation of the weak Scalability of Sigmoid algorithm using Gustafson Law.

## 6. Related work

The development of heterogeneous systems and their advantages in performance and energy consumption, has led to great interest from industry and academia. However, despite all the efforts, the problem of adequately using the enormous computing capabilities of this kind of systems is to be solved. Many research papers have addressed it from two angles: task and data parallelism.

The idea behind task parallelism is distributing independent tasks to each of the available devices, so ideally their resources fully used. Such is the approach of [16], which proposes a lightweight runtime based on QUARK, that distributes tasks using task superscalar scheduling and a greedy heuristic. The authors of [41] apply fuzzy neural networks to the task distribution problem. MultiCL [2] is an OpenCL runtime for task-parallel workloads based on storing execution information for each kernel-device pair and using it for future kernel launches. VirtCL is an OpenCL system abstraction framework [40] that implements a history-based task scheduler that constructs regression models to predict turnaround times. The authors of [12] propose SPARTA, a throughput-aware runtime task allocator for Heterogeneous Many Core platforms [12]. It analyzes tasks at runtime and uses the information to maximize energy-efficiency. Unicorn is a parallel programming model for hybrid CPU-GPU clusters that implements a dynamic work-stealing task scheduler [4]. Another example of the work-stealing approach is that of Xkaapi [14].

All these works are remarkable efforts towards the efficient use of the capabilities offered by heterogeneous systems. However, these techniques are often not enough. Task-parallelism is of little use when there are few kernels or they have dependencies. This is the case of many current workloads that use very big data sets, such as machine learning applications. In these cases, the only way to extract parallelism is for all the devices to collaborate in the execution of a single task, via data-parallelism. This is when co-execution comes into play, which is the focus of Sigmoid.

Considering this, MKMD is an intermediate approach [22]. It combines coarse grain scheduling of indivisible kernels, followed by opportunistic fine grained work-group distribution if idle slots are detected. To do so, it predicts the execution time of work-groups and uses a heuristic. The work of Zhong et al. focuses on splitting different tasks to maximise the usage of a single GPU. This could be extended to several devices by adding an extra dispatcher [43].

Several papers have tackled load balancing for data parallelism. As substantiated throughout this paper, for a single load balancing technique to succeed, it has to be flexible and consider both the heterogeneity of the devices and the irregularity of the applications. Additionally, it should not require excessive information from the programmer.

Kim et al. [18] approach the problem by implementing an OpenCL framework that provides a view of a single compute device by transparently managing the memory of the devices. Their approach to data-parallelism is based on a static load balancing strategy, so it cannot adapt to irregularity. Besides, they only consider systems with several identical GPUs and ignore CPUs, so their proposal is not suitable for truly heterogeneous systems. Lee et al.[21] propose automatic modifications to OpenCL code that executes on a single device, so the load is balanced among several ones. De la Lama et al. [11] propose a library that implements static load balancing by encapsulating standard OpenCL calls. The work presented in [20] uses machine learning techniques to build an offline model that predicts an ideal static load partitioning. However, this model does not consider irregularity. Similarly, Zhong et al. [44] use performance models to identify an ideal static work distribution. In [23] the focus is on the static distribution of a kernel execution to the available devices via code modifications.

Other authors have proposed training-based load balancing methods. Qilin [25] proposes the use of a execution-time database for all the programs the system has executed and a linear regression model. Another such approach is that of Maestro [37]. These techniques are only useful in systems that run the same applications frequently.

All the above works propose static approaches, so they fail to address the importance of adaptiveness. Moreover, even when facing regular workloads, these methods cannot recover from an inaccurate initial estimation of the capabilities of the available devices. Dynamic methods, such as Sigmoid, try to prevent these issues by dividing the workload in smaller chunks and taking load balancing decisions at runtime.

FluidicCL [32] implements an adaptive dynamic scheduler, but only focuses on systems with one CPU and one GPU. SnuCL [19] is an OpenCL framework for heterogeneous CPU/GPU clusters. However, regarding load balancing, it is capable of either dynamically distributing a kernel among the CPU cores or using a GPU, but does not support the co-execution of a kernel.

Kaleem et al. in [17] and Boyer et al. in [6] propose adaptive methods that use the execution time of the first packages to distribute the load. However, they focus on a CPU/GPU scenario and, unlike Sigmoid, do not scale well to configurations with more devices. Similarly, HDSS [3] is a load balancing algorithm that dynamically learns the computational speed of each processor and then schedules the remainder of the workload using a weighted self-scheduling scheme. However, this algorithm assumes that the packages launched in the initial phase are representative of the whole load, which might not be true for irregular kernels. Besides, package size decreases linearly during the completion phase, which may produce unnecessary overheads as substantiated in this paper. Navarro et al. [27] propose a dynamic, adaptive algorithm for Threading Building Blocks (TBB) that uses a fixed package size for the GPU and a variable one for the CPU. This work was extended in [28], proposing an adaptive package size for the GPU too. This is also based on using small initial packages to identify a package size that obtains near optimal performance.

Scogland et al. [36] propose several distribution schemes that fit accelerated OpenMP computing patterns. However, they do not propose a single solution to the load balancing problem. The library presented in [33] also implements several load balancing algorithms and proposes the HGuided, which adapts to irregularity and heterogeneity. However, it requires certain parameters from the programmer and uses linearly decreasing packages that might impose overheads.

Finepar [42] is a software that analyzes an irregular application, its input data and the available (integrated) hardware and builds a performance model to obtain an ideal work partition. The model calculation has to be performed every time the input data changes, which is costly. This software only supports applications based on sparse matrices.

Finally, some papers propose algorithms to distribute the workload accounting for performance and power. GreenGPU dynamically distributes work to GPU and CPU, minimizing the energy wasted on idling and waiting for the slowest device [26]. To maximise energy savings while avoiding performance degradation, it throttles the frequencies of CPU, GPU and memory, based on their utilisation. Wang and Ren [39] propose several analytical models and guidelines to produce partitions balance performance and energy consumption.

## 7. Conclusion

This paper presents Sigmoid, a new load balancing algorithm. It allows executing a single data-parallel OpenCL kernel taking full advantage of all the compute devices in a heterogeneous system. This algorithm is dynamic, as it distributes the workload among the devices at run time. It is also adaptive, since it can change its behaviour during the execution to adapt to regular and irregular kernels. Additionally it starts with large packages to reduce the synchronisation points at the beginning, reducing overheads. As execution advances, it shortens the package size to increase the granularity toward the end, allowing perfect load balance. Finally, using all these features requires very limited effort, since Sigmoid is controlled by a small amount of settings, which can be easily established at installation time.

The Sigmoid algorithm has been implemented in Maat, an OpenCL library that simplifies the management of heterogeneous systems when co-executing data-parallel kernels. An exhaustive experimental evaluation has confirmed that the use of the whole heterogeneous system improves performance and energy consumption when compared to using the devices individually. The experimental results show that Sigmoid reaches an almost perfect load balance, with kernels of diverse behaviours. As a consequence it gives the best performance and energy consumption results compared to the other evaluated alternatives for most of the benchmarks considered in this work. Additionally, it has proven excellent scalability properties. Sigmoid delivers these good results through adaptation, using few parameters set at installation time, without the need of any further time-consuming parameter search. When compared to other effortless load balancing algorithms, Sigmoid delivers better results, both regarding performance and energy consumption.

In the future, Sigmoid will be tested with real parallel applications as well as in other kinds of heterogeneous systems composed of FPGAs and tensor processing units.

## CRediT authorship contribution statement

**Borja Pérez:** Conceptualization, Investigation, Software, Validation, Writing – original draft. **E. Stafford:** Investigation, Software, Visualization, Writing – original draft. **J.L. Bosque:** Conceptualization, Investigation, Methodology, Writing – original draft. **R. Beivide:** Conceptualization, Funding acquisition, Supervision.

## Declaration of competing interest

The authors have no known conflicts of interest beyond the ones related to organisations they belong to, which are Universidad de Cantabria and Barcelona Supercomputing Center.

## Acknowledgment

This work has been supported by the Spanish Science and Technology Commission under contract PID2019-105660RB-C22 and the European HiPEAC Network of Excellence.

## References

[1] A. Acosta, R. Corujo, V. Blanco, F. Almeida, Dynamic load balancing on heterogeneous multicore/multiGPU systems, in: W.W. Smari, J.P. McIntire (Eds.), HPCS, 2010, pp. 467–476.
[2] A.M. Aji, A.J. Peña, P. Balaji, W.-c. Feng Multicl, Parallel Comput. 58 (C) (2016) 37–55.
[3] M.E. Belviranli, L.N. Bhuyan, R. Gupta, A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures, ACM Trans. Archit. Code Optim. 9 (4) (2013) 1–20.
[4] T. Beri, S. Bansal, S. Kumar, The unicorn runtime: efficient distributed shared memory programming for hybrid cpu-gpu clusters, IEEE Trans. Parallel Distrib. Syst. 28 (5) (2017) 1518–1534.
[5] C.M. Bishop, Pattern Recognition and Machine Learning, Information Science and Statistics, Springer-Verlag, Secaucus, 2006.
[6] M. Boyer, K. Skadron, S. Che, N. Jayasena, Load balancing in a changing world: dealing with heterogeneity and performance variability, in: ACM Int. Computing Frontiers, 2013, pp. 1–10.
[7] J. Cabezas, I. Gelado, J.E. Stone, N. Navarro, D.B. Kirk, W.m. Hwu, Runtime and architecture support for efficient data exchange in multi-accelerator applications, IEEE Trans. Parallel Distrib. Syst. 26 (5) (2015) 1405–1418.
[8] E. Castillo, C. Camarero, A. Borrego, J.L. Bosque, Financial applications on multi-cpu and multi-gpu architectures, J. Supercomput. 71 (2) (2015) 729–739.
[9] B. Chapman, G. Jost, R.v.d. Pas, Using OpenMP: Portable Shared Memory Parallel Programming, The MIT Press, 2007.
[10] K. Dabov, A. Foi, V. Katkovnik, K. Egiazarian, Image denoising by sparse 3-d transform-domain collaborative filtering, IEEE Trans. Image Process. 16 (8) (2007) 2080–2095.
[11] C.S. de la Lama, P. Toharia, J.L. Bosque, O.D. Robles, Static multi-device load balancing for opencl, in: Proc. of ISPA, IEEE Computer Society, 2012, pp. 675–682.
[12] B. Donyanavard, T. Mück, S. Sarma, N. Dutt Sparta, Runtime task allocation for energy efficient heterogeneous many-cores, in: Proc. of the 11th IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis, CODES '16, 2016, pp. 1–10.
[13] S. Dreiseitl, L. Ohno-Machado, Logistic regression and artificial neural network classification models: a methodology review, J. Biomed. Inform. 35 (5) (2002) 352–359.
[14] T. Gautier, J. Lima, N. Maillard, B. Raffin, Xkaapi: a runtime system for data-flow task programming on heterogeneous architectures, in: Proc. of IPDPS, 2013, pp. 1299–1308.
[15] J.L. Gustafson, Reevaluating Amdahl's law, Commun. ACM 31 (5) (1988) 532–533.
[16] A. Haidar, C. Cao, A. Yarkhan, P. Luszczek, S. Tomov, K. Kabir, J. Dongarra, Unified development for mixed multi-GPU and multi-coprocessor environments using a lightweight runtime environment, in: Proc. of IPDPS, 2014, pp. 491–500.
[17] R. Kaleem, R. Barik, T. Shpeisman, B.T. Lewis, C. Hu, K. Pingali, Adaptive heterogeneous scheduling for integrated GPUs, in: Proc. of PACT, 2014, pp. 151–162.
[18] J. Kim, H. Kim, J. Lee, J. Lee, Achieving a single compute device image in OpenCL for multiple GPUs, in: Proc. of the ACM PPoPP, ACM, 2011, pp. 277–287.
[19] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, J. Lee, SnuCL: an opencl framework for heterogeneous CPU/GPU clusters, in: Proceedings of the ACM ICS, 2012, pp. 341–352.
[20] K. Kofler, I. Grasso, B. Cosenza, T. Fahringer, An automatic input-sensitive approach for heterogeneous task partitioning, in: Proc. of the 27th Int. ACM Conf. on Supercomputing, ICS, 2013.
[21] J. Lee, M. Samadi, Y. Park, S. Mahlke, Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems, in: Proc. of PACT, 2013, pp. 245–256.
[22] J. Lee, M. Samadi, S. Mahlke, Orchestrating multiple data-parallel kernels on multiple devices, in: Int. Conf. on Parallel Architecture and Compilation (PACT), 2015, pp. 355–366.
[23] J. Lee, M. Samadi, Y. Park, S. Mahlke, Skmd: single kernel on multiple devices for transparent cpu-gpu collaboration, ACM Trans. Comput. Syst. 33 (3) (2015) 9.
[24] C. Lin, C. Liu, L. Chien, S. Chang, Accelerating pattern matching using a novel parallel algorithm on gpus, IEEE Trans. Comput. 62 (10) (2013) 1906–1916.
[25] C.-K. Luk, S. Hong, H. Kim Qilin, Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping, in: Proc. of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture, in: MICRO, vol. 42, 2009, pp. 45–55.
[26] K. Ma, Y. Bai, X. Wang, W. Chen, X. Li, Energy conservation for GPU–CPU architectures with dynamic workload division and frequency scaling, Sustain. Comput. Inform. Syst. 12 (2016) 21–33.
[27] A. Navarro, A. Vilches, F. Corbera, R. Asenjo, Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures, J. Supercomput. 70 (2) (2014) 756–771.
[28] A. Navarro, F. Corbera, A. Rodríguez, A. Vilches, R. Asenjo, Heterogeneous parallel_for template for CPU-GPU chips, Int. J. Parallel Program. 47 (2019) 213–233.

[29] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with cuda, Queue 6 (2) (2008) 40–53.
[30] NVIDIA, Kepler GK110 whitepaper, http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.
[31] NVIDIA, NVIDIA management library (NVML), https://developer.nvidia.com/nvidia-management-library-nvml, 2018. (Accessed May 2018).
[32] P. Pandit, R. Govindarajan, Fluidic kernels: cooperative execution of opencl programs on multiple heterogeneous devices, in: Proceedings of Annual IEEE/ACM CGO, 2014, pp. 273–283.
[33] B. Pérez, J.L. Bosque, R. Beivide, Simplifying programming and load balancing of data parallel applications on heterogeneous systems, in: Proc. of the 9th ACM Work. on General Purpose Processing Using Graphics Processing Unit, GPGPU '16, 2016, pp. 42–51.
[34] B. Pérez, E. Stafford, J.L. Bosque, R. Beivide, Energy efficiency of load balancing for data-parallel applications in heterogeneous systems, J. Supercomput. 73 (2017) 330–342.
[35] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, E. Weissmann, Power management architecture of the 2nd generation Intel Core microarchitecture, formerly codenamed Sandy Bridge, in: IEEE Int. HotChips Symp. on High-Perf. Chips, 2011.
[36] T. Scogland, B. Rountree, W. chun Feng, B. de Supinski, Heterogeneous task scheduling for accelerated openmp, in: Proc. IPDPS, 2012, pp. 144–155.
[37] K. Spafford, J. Meredith, J. Vetter Maestro, Data orchestration and tuning for opencl devices, in: Proc. of the 16th Int. Euro-Par Conf. on Parallel Processing, Euro-Par'10, 2010, pp. 275–286.
[38] J.E. Stone, D. Gohara, G. Shi, OpenCL: a parallel programming standard for heterogeneous computing systems, IEEE Des. Test Comput. 12 (3) (2010) 66–73.
[39] L. Tang, R.F. Barrett, J. Cook, X.S. Hu, Peapaw: performance and energy-aware partitioning of workload on heterogeneous platforms, ACM Trans. Des. Autom. Electron. Syst. 22 (3) (2017) 41.
[40] Y.-P. You, H.-J. Wu, Y.-N. Tsai, Y.-T. Chao Virtcl, A framework for OpenCL device abstraction and management, in: Principles and Practice of Parallel Programming, PPoPP 2015, ACM, 2015.
[41] C. Zhang, Y. Xu, J. Zhou, Z. Xu, L. Lu, J. Lu, Dynamic load balancing on multi-gpus system for big data processing, in: 23rd Int. Conf. on Automation and Computing (ICAC), 2017, pp. 1–6.
[42] F. Zhang, B. Wu, J. Zhai, B. He, W. Chen, Finepar: irregularity-aware fine-grained workload partitioning on integrated architectures, in: 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2017, pp. 27–38.
[43] J. Zhong, B. He, Kernelet: high-throughput GPU kernel executions with dynamic slicing and scheduling, CoRR, arXiv:1303.5164 [abs], 2013, pp. 1522–1532.
[44] Z. Zhong, V. Rychkov, A. Lastovetsky, Data partitioning on multicore and multi-gpu platforms using functional performance models, IEEE Trans. Comput. 64 (9) (2015) 2506–2518.

**Borja Pérez** graduated in Computer Science from University of Cantabria in 2014 and is currently pursuing a PhD degree on Technology and Science. He is a Pre-PhD researcher in the Dept. of Computer Engineering and Electronics of the University of Cantabria. His research interests include heterogeneous sytems both from the architecture and the programming view point and high performance computing.

**Esteban Stafford** received the M. Sc. degree in Telecommunication Engineering from the University of Cantabria in 2001 and he obtained the PhD degree in Computer Science in 2015. He is currently a part-time professor in the Dept. of Computer Engineering and Electronics of the University of Cantabria. His research interests include computer architecture, parallel computers and interconnection networks.

**Jose Luis Bosque** graduated in Computer Science from Universidad PolitÃĬcnica de Madrid in 1994. He received the PhD degree in Computer Science and Engineering in 2003 and the Extraordinary Ph.D Award from the same University. He joined the Universidad de Cantabria in 2006, where he is currently Associate Professor in the Department of Computer and Electronics. His research interests include high performance computing, heterogeneous systems and interconnection networks.

**Ramón Beivide** received the B.Sc. and M.Sc. degrees in Computer Science from the Universidad Autónoma de Barcelona (UAB) in 1981 and 1982. The Ph.D. degree, also in Computer Science, from the Universidad Politécnica de Catalunya (UPC) in 1985. He joined the Universidad de Cantabria in 1991, where he is currently a Professor in the Dept. of Computer Engineering and Electronics. His research interests include computer architecture, interconnection networks, coding theory and graph theory.